CAD Tool Interchangeability through Net List Translation

Steve Meyer

Pragmatic C Software
220 Montgomery Street. Suite 925
San Francisco, CA 94104

Abstract: The argument for electronic circuit logic design tool interchangeability by means of source-destination specific net list translation within a framework of Unix operating system commands is presented. After describing the information necessary for design verification using an assortment of diverse tools, various design tool interchangeability alternatives are considered. Discussion of net list translator examples, Unix like design movement programs, and translation speed complete the argument.

1. Introduction

The number of available electronic circuit logic design and verification tools has increased in recent years. This tool availability has created a problem in tool interchangeability. From the circuit designer's viewpoint, there is great advantage in the ability to mix and match ones favorite tools. It is also valuable to be able to use tools with special capabilities even though such tools may not be general purpose enough to replace, say, a company wide standard simulator. The advantages of tool interchangeability have boon shown in various case studies. See [22] for an example in which two gate level simulators were used to verify a gate array design. Each simulator found timing problems missed by the other. This trend toward multiple design tools has recently been acknowledged by established CAD tool providers in what they call open systems or frameworks.

Most recent electronic circuit design methodology papers argue that design should be accomplished at a higher level, and that designers should be shielded from design verification tool details (see [3] for example). According to this view, not only design objects but also tools should be opaque objects within an object oriented design framework. This paper expresses the contrary viewpoint. It argues that logic design should move to a lower, more design specific, more tool specific, more concrete, and more accurate level. According to the view expressed here, the widely available Unix operating system [23] [2] provides as much framework as necessary. Source tool to destination tool net list language translation is then used for design movement between tools. Data is stored in normal ascii or binary files and maintained with Unix framework commands (cf. make [8] and SCCS [1][26][2. vol. 3]).

Net list translation plays the same role in design verification that Unix filters play in Unix text processing. Translators are filters with optional configuration and mapping libraries to control the filtering. Translation is used to move designs between the various verification tools classes: schematic editors. timing calculators, simulators. timing verifies. and layout systems. This paper can alternatively be viewed as a commentary on various CAD opportunities and problems that arise from net list translation development.

The most obvious scheme for moving electronic design descriptions from the language format required by one tool to another tool's format is by means of a computer program. That computer program is called a net list translator. Since net list translators are low level tools that require careful attention to detail and considerable development effort, various alternative approaches to the tool interchangeability problem have been proposed. Before discussing the various alternative proposals, it is necessary to define the information required to characterize an electronic circuit and to describe the net list translation task.

2. The Four Kinds of Design Information

The main circuit description is the net list itself. It specifies cell-port-net connections, I/O ports, and primitive logic function types used in a circuit. Test patterns describe circuit behavior from one viewpoint and are the legal circuit definition when a circuit is moved to an ASIC vendor. Behavioral models and programs describe circuit behavior from another viewpoint. The fourth information class includes primitive type element definitions (called models or model libraries). Primitives can be macro cells for ASIC designs, TTL parts for PCB designs, or transistor models for custom design. Since primitive type models are either represented as behavioral programs or net lists that are expanded before tool allocation, this fourth class either fits into the net list or the behavioral information categories and will not be considered separately.

Test pattern movement from one tool to another is usually easy and can be accomplished with an editor or macro language script. There is currently no general solution to behavioral module translation from one language into another since the problem is no easier than translating from one computer programming language into another. This requirement for manual recoding is usually not serious since before manufacture, any hardware system must be decomposed into a primitive based net list containing no behavioral descriptions. The net list form rather than the behavioral form component model description can be moved to the new tool. In early design stages when only behavioral models exist, it makes no sense to apply tools that do not support behavioral circuit description. Another use for behavior modeling is in test scaffolding and circuit debugging. In these cases one would expect to redo the behavioral descriptions for the new tool since the point to moving to a new tool is to take advantage of its different capabilities. The different test scaffolding can help to test the assumptions made in the original testing and debugging programs. Another possibility is to execute the behavioral test scaffolding on the original system while monitoring I/O ports. The sampled values are used as test pattern to drive the destination tool.

Notice this comprehensive design information movement paradigm allows all destination tool features to be utilized. If the object approach were used, only circuit description elements already built into the black box framework could be utilized by the destination tool. Of course, conventions and computer programs such as those supplied by Unix that improve user interface consistency and ease of learning are valuable.

3. Net List Translator Functional Requirements

A net list translator must preserve circuit meaning so that both structural gate level connectivity and semantic function are identical on the systems providing the target system implements the necessary connection pattern functionality. When the source and destination function do not correspond, ahe translation needs to generate a net list that will have equivalent functionality for the destination tool's intended application. When construct mapping is ambiguous, a means for the user to specify translator mapping is required the target system implements the necessary connection pattern functionality for the destination tool's intended application. When construct mapping is ambiguous, a means for the user to specify translator mapping is required. Optional translation libraries need to be provided to allow different translation levels such as exact timing versus unit delay timing and to allow primitive name mapping in order to facilitate destination net list use with pre-existing libraries (precoded models). A translator should support all conceptual design aids: vectored wires (busses), signal wire concatenations, vectored part instances, synonyms, strength modeling, automatic name conflict repair, property mapping, timing checks, and even perhaps minor rewiring. A translator should map any feature that does not correspond exactly in the destination format into something as close as possible using the same coding style. A translator should never translate something that will have different destination tool semantic meaning unless errors or warnings are emitted.

4. Current Translator Limitations

One reason for skepticism toward net list translators is that many current translators are not complete. Many translators are just net-port or cell-port connection reformatters. They often use an automatic parsing system such as the lex/yacc combination [15][12]. This method is almost always unable to deal with the context sensitive nature of net lists and can not translate commonly occurring formats in which a primitive's type determines not only syntactic form but also lexical structure. For example, the entire TDL language [28] requires a part type recognition pass before the DEFINE section instances can be parsed. Hilo has various primitives such as capacitor that change element parameter interpretation. Language specific recursive descent (usually without any need for recursion) is advantageous for translating such irregular formats. Parsing routines can be passed the current type and then use it to determine tokenization. It is not well understood that there is more to gate level circuit translation than simple net connection translation. Translation to insure identical cell-port-net and net-port-cell connections is not sufficient. Circuit descriptions also contain implicit semantic information that is built into the source tool verification algorithm and source language models. For example, some simulators model mos style gates by building strengths into the primitive models (see [18] for example) while others use general purpose primitives for which each instantiation must specify the required rise and fall strengths (see [10] for example). For output instances the mapped to type may need to include various instance parameters added by the translator.

Many net list translators handle special source language features by producing a destination format net list that is syntactically correct but expresses the wrong meaning. This is worse than not emitting anything since when a designer finds a problem during destination tool use, problem location is difficult. For example, since the Valid system simulator tracks and stores both logic polarity rails, net lists use a property (called BUBBLED) to indicate which rail to select during simulation. For destination tools that do not support dual rail logic, it is necessary to add inverters. Translators that do not add the extra logic cause incorrect destination tool results. The LSI Logic TDL dialect called NDL [17] allows modules that have outputs that are effectively power supply generators. If translated modules do not have output I/O ports connected to the supply net, when the module is expanded (flattened) before simulation, the higher level signal nets will not be correctly tied to the power supply. Some output formats allow outputs to be tied to power supplies while others require logic gate addition.

5. Net List Translation Alternatives

Before considering the various possible disadvantages of net list translation, the adoption of one universal design standard is considered. Standardization eliminates any need for net list translation because all electronic circuits are stored in the one standard format. This section concludes by considering two proposals that while still requiring net list translation, claim to simplify tool interchangeability. One proposal would standardize on one universal intermediate language. Net list translations would then be written to translate into and out of the standard form. The other proposal uses net list translation within a tool framework. The tool framework then supposedly simplifies circuit design through abstraction and detail hiding.

5.1 One Universal Standard

If one universal standard could be agreed upon, it would solve the tool interchangeability problem by standardizing it out of existence. CAD tool development would become implementation of the one standard tool set The main problem with universal standards is that it is virtually impossible in areas of design methodology to achieve the consensus necessary for standard adoption. See for example the proposed Japanese alternative to VHDL discussed at the 1989 Design Automation Conference [30]. Even in the unlikely event that one design verification standard could be agreed upon, there are still inherent problems with attempting to standardize in areas involving human problem solving. At first glance, a universal standard seems to have numerous advantages such as elimination of designer retraining and model recoding. In practice, any standard tool set modeling weakness would cause errors associated with the weakness to appear in every circuit using the technology for which the weakness applies. Also, standardization would stop progress in design tool development. There would be no possibility of discovery outside the assumptions of the standard. The only possible progress would be improvements in simulation speed or design representation implementations of the one standard tool set. It is not difficult to imagine potential large improvements in current design methods and tools. What would happen if someone discovers a new conceptual framework for circuit simulation that is as accurate as spice but allows simulations to run as fast as current gate level simulations? There would be no way to test it. It could not be developed, and its adoption as the new standard would be prohibitively difficult. The current primary standardization candidate, VHDL [7][16][4] which defines both verification semantics and language format has potential difficulties. VHDL is based on another standard (the ADA programming language) that itself is not being adopted as rapidly as had been hoped. Two additional problems were mentioned at the 1989 DAC panel on standardization [9]. The view that programming language abstract data types were not appropriate for hardware design was expressed. If hardware design is programming, why bother building circuits, just write microprocessor programs. Problems with the VHDL timing model were mentioned. VHDL proposes the unlikely combination of a digital gate timing model with programming language task semantics. The standard precludes a design group from, for example, preferring the traditional system simulation style task independent time token movement model. VHDL makes sense as a design environment alternative and is worth developing if for no other reason than the popularity of its data type model in computer programming.

5.2 Source-Destination Specific Translation

Given that net list translators correctly map all features and semantic meaning such that designers trust the destination net list format veracity (see section 3), they are a workable solution to tool interchangeability. The main objection to net list translators has been the seemingly unbounded effort required in implementing one translator for every source-destination format combination. In practice, required programming effort is not a problem. There are only around 5 to 10 widely used formats. No one organization needs to provide translators between every combination. Any given tool provider only needs to provide "into" translators for its net list format or formats. All tool providers taken together provide the complete translator set. Programming effort can be minimized by using source code from a related finished translator as the stem or template for new translators. Translation from one source to one destination format is concrete and well understood. Of course, translator programming requires experienced programmers since tools with large user communities always seem to contain numerous complicated special purpose features. The expertise required to handle large programs and to understand net list features that are similar but not identical across different translators must be present. Treating program routines as opaque objects is probably not sufficient for the subtle semantic differences between formats.

5.3 One Universal Intermediate Form

Since it is unlikely that a universal standard will be adopted, another approach to tool interchangeability retains net list translation but attempts to simplify the translation task by using one common intermediate form. The EDIF [6][25] design interchange format is currently the main example of this approach. The possible advantage is that for each tool specific net list language only an "into" and an "out of" translator are needed. This idea has practical problems. Since the standard format must be completely general, it will handle the details of no format exactly, and even very close languages such as TDL [28] and its NDL dialect [17] become as difficult to translate as completely unrelated formats. The mapping problem that involves translating from a general intermediate language whose original source net list language is unknown into a specific format is difficult. What happens is that the syntax (lisp s-expressions for EDIF) stays universal but properties are created that are added by the "into" translator and by convention recognized and processed by the "out of" translator. Since any standard must try to fit every case, no matter how comprehensive such a standard is, there will be cases that do not fit. The net list view part of EDIF is not comprehensive enough to represent all net list features. This causes the EDIF property and userdata constructs ([6], p. 2-313, 2-383) to be used with the effect of moving back to the source-destination specific net list translator scheme. Many net list languages support high to low and low to high bus ranges that do not start at zero, but EDIF arrays are unidirectional and always start at zero

([6], p. 2-11). Therefore some property is required. EDIF defines a simple delay calculator model ([6], 2-295, 2-69 through 2-72), but, at least in the case of semiconductor vendor ASIC libraries, most timing algorithms do not fit the EDIF model. The EDIF signal strength scheme allows a strength order relation ([6], p. 2-87, 2-356, 2-405), but this is a mere shadow of the complicated semantic and behavioral meaning of strengths in real simulators.

5.4 Frameworks

Frameworks do not solve the tool interchangeability problem by themselves since within the framework, either a universal standard or net list translation are required. Given that net list translation is still required, frameworks are valuable to the extent they provide conventions and shells to simplify learning and using tool interfaces. As long as the standard is flexible enough to allow both tool and design growth, frameworks are positive. Frameworks are detrimental to circuit design if they express the modern tendency to believe design problems are solved by simply providing a meta level design environment. Systems that implement abstraction and detail hiding often result in designs for which details are not well handled. If a framework is too narrow and standardized, it precludes utilization of a tool's special features, and any framework that makes design verification model assumptions (such as the VHDL or EDIF standards) will probably be narrow in exactly this sense. For example, a tool's ability to handle general signal name synonyms requires that the framework have a synonym capability, but then that complicated and probably slow synonym mechanism must be present even for tools that do not use synonyms. The Unix operating system itself may be the best CAD tool framework. Its computer interface model has gained wide acceptance during twenty years of use. It allows individual design verification tools to use the most appropriate net list formats and interfaces. [27] describes the Vivid system framework implementation based on Unix (see also [11][19]).

6. Examples

There are certain location within the electronic system design cycle at which the most benefit from alternative or supplementary tools can be gained. This section discusses net list translation at those points.

6.1 Graphic Editor Output Translation

Since schematic editor choice depends upon widely varying individual taste, it is possible for a project or company to mix a schematic editor from one tool set with verification tools from another. There are two common approaches to schematic level translation. One translates the raw schematic editor output which is usually expressed as geometric drawing commands. This approach has the advantage that it requires no intermediate processing on the source graphics editor system, but has the disadvantage that it is difficult to reconstruct the implicit semantic data that are assumed by the schematic editor. Examples axe unexpanded macros, multiple fan-in signal nets, and properties that are removed by the first stage circuit compiler. Any translator that translates raw schematic data must effectively implement the source system's schematic interpretation algorithm. Circuit compiler output is another entry point. This has the disadvantage that the source system circuit compiler may be slow but has the advantage that the output is in net list format and that the format is usually well defined. Examples are the NDL dialect of TDL that is output by the LSED schematic editor [17] and the Valid cmpexp.dat file [18]. If the source tool compiler is used to additionally expand a circuit down to built in primitives, a new tool can be used after coding only those few primitives in the destination tool format. After more models are available, unexpanded source drawings can be used to move hierarchical modules to the destination tool.

6.2 Translation at the Manufacturing Interface

Since net lists are the normal entry point to a manufacturing system, at least for ASIC vendors, any circuit that is not designed on the vendor's system requires translation into the vendor's verification tool format. Completed circuits can be translated a second time to move the circuit back into the original design system. The second translation both assures that the manufactured circuit exactly matches the original circuit and allows further verification at the system level. Semiconductor interface net list translations are among the easiest since the vendor requires that circuits be limited to its general purpose macro cells (mask patterns need to exist), and does not require timing translation since timing is built into the vendor's sign off simulation system. Translation back from a vendor's system is frequently more difficult since timing information including exact wire delays needs to be moved.

6.3 Translation for Mixed Gate and Behavioral Verification

Development of verification systems that combine behavioral and gate level simulation has increased the potential benefits from net list translation. Net list translators can improve system simulation speed. Simulators that mix accurate time modeling including timing checks and delay calculator delays, unit delay modeling, behavioral simulation, and programming language system control and debugging allow fast and accurate system level verification. In one verification method, net list translation is used twice. Initially, unfinished parts or even subsystems are modeled behaviorally. The behavioral model may only implement the circuit's bus interface protocol or it may model complete functionality with a program. Once the gate level representation is completed, perhaps using some automated method, the previous incomplete behavioral model is replaced by the completed gate model. Usually the completed model will be the signed off model translated back from the semiconductor vendor's system. Since by the time a part is ready for fabrication, complete system level tests would exist, those tests can be rerun with the previous behavioral model replaced by the exact timing gate model. The simulation will run more slowly but any problems in the gate level implementation can be found and corrected. The most common problems found at this point are timing problems and function misunderstanding between system and IC designers. For timing critical parts such as vector pipelines and bus interfaces, it may be necessary to run simulations with various combinations of behavioral and gate model instances in various locations to detect subtle timing interaction problems. Once system function is verified using exact timing, the same part can then be retranslated using unit delays so large system level simulation will run faster. A unit delay gate model with a large percentage of random logic will usually simulate faster than a behavioral model that implements full functionality. In fact, software simulation on a general purpose work station using mixed model timing detail can simulate faster than hardware simulators that use exact timing gate models for an entire system. The mixed level software method achieves at least as accurate timing and allows much better monitoring and debugging. Finally, behavioral, unit delay, and "real" delay simulation together can find Circuit problems that one simulation method alone would not find.

6.4 Translation to Reduce the Tool Change Learning Curve

When an organization decides to change design tools and design methodology, the change over process can be traumatic. If net list translators are available to move old designs from the previous format into the new one and to move circuits back from the new format to the old, the transition can be made less difficult. For experienced designers the ability to work with familiar circuits simplifies learning the new system. Inexperienced designers can use the capability to move circuits from the new tool to the old to make a small design step and then move back to familiar ground to assure the small design step was correct. This has great psychological benefit for designers reticent about making the plunge into a new environment all at once.

6.5 Translation to Change Model Packaging

When a net list translation step is included in a circuit development edit-compile-translate-simulate loop, the translate step can be used to control model packaging and net list style within the destination tool. Translator configuration and mapping library files are normally used to map names (i.e. equivalent models named differently) and to allow user translation feature control for ambiguous or not exactly mapable features, but translation can also be used to change output style. Instance and vector wire splitting versus recombination can change destination net list part packaging. Parts can be removed from the output net list. Related instances can be concatenated into one instance with a different type for which the I/O ports from each instance are concatenated into one large instance I/O port list. Type or instance specific delays can be selected. One design group wanted the capability to leave certain vector widths unsplit but to split others depending on instance vector size since the different sizes were modeled differently.

7. Unix Framework Tools For Net List Translation

Various programs the are similar in function to Unix commands can be used in conjunction with net list translators to improve circuit design. The complicated output mapping described in section 6.5 can benefit from mapping library maintenance tools. One such tool builds library templates from source format net lists. Since translators can automatically determine type elements I/O ports from the input net list, translation libraries are optional, but the ability to start from default map templates simplifies map library preparation for those parts that need special mapping. Another useful utility is a library difference finder that is library syntax and semantics specific. It is like the Unix diff program but specialized to a given translator library format and can therefore deal with different orderings and codings that are equivalent in effect. The library difference finding program usually also has lint [13] like consistency checks. Other programs are possible. A program to build library map entries for TTL part libraries on different CAD systems that were originally coded from different data book versions is useful. It reads both the source and destination language model I/O port lists and produces a translator map library that maps the port names. A good algorithm uses a dynamic programming string to string mapping algorithm to match ports [29]. The port name matching problem is easier than matching natural language words since port names have no related letter combinations and frequencies [21]. Utilities that generate make program make files [8] which automatically execute circuit compilation and net list translation can be written. Normally, make does not work well when rules for making things can not be expressed as file name suffix transformations. For example on the Valid system, a program can read the drawing hierarchy and determines design true dependencies. The program then writes a make file containing usually hundreds of literal rules, one rule for each module type. The make file generator utility can be run after each schematic editor change for completely automatic translation, but it only needs to be run if new user module or library types are added to some drawing. See the Vivid system description for a program that builds a make file from a manually prepared design description script [27]. The utility speeds up simulation preparation time since small circuit changes require only local translation and probably also local compilation.

8. Translation Speed

If an extra net list translation step adds significant time to the edit-simulate loop, it would reduce the advantages of tool interchangeability through net list translation. A net list translation step normally takes about twice the time it takes to read and tokenize the input net list language plus the disk write time to write the output format (see [20][for translation speed measurements). This duration is enough faster than even good circuit compilation (expansion or flattening) programs so that a translation step is usually not a noticeable time factor in the edit-simulate cycle. See [5] for circuit expansion speed measurements.

Recent work by Jones [14] attempts to show that circuit compilation can be executed must faster than has been thought. If this is true, the extra time used by a net list translation step would become a more significant factor. I believe those results have the following three flaws which suggest circuit compilation can not be significantly speeded up without functionality loss. The speed measure presented was the number of different modules (types) used plus the number of after expansion nodes (instances) per unit time ([14], section 6). Since this measure is I/O port number independent, it does not seem to correlate with expansion time for real circuits. Certainly an expansion of a circuit containing only modules with hundreds of I/O ports will be significantly slower than one in which all modules have less than 10 I/O ports. Since the schematic net list storage form used by the possibly faster method uses a "connected to" relation that is searched linearly and attached to the module type (p. 823), the nodes in the measured circuits are probably small. Second, all results assume no disk I/O. If in memory only operation were feasible, it would speed up processing, but some circuit description form needs to be stored on disk. If all in memory processing were practical, net list translation and other flattening algorithms would be speeded up by an equal amount. Third, the circuit model is so simplified that it may not be useful in practice. The extra details would probably make the DAG representation scheme unusable (p. 824). There is no fully qualified name storage. Imagine trying to debug with a simulator for which each net and instance lacks any name hierarchy but is identified by only a number. There is no multiple fan in (wired) net handling. This would at least require a more complicated DAG node connectivity representation scheme. There is no way to assign instance specific properties. This means no user assigned instance names. There is no way to traverse the net list to determine connectivity and fan-out for delay calculators without repeated relation searches.

There are actually situations where net list translation can reduce the total time needed to prepare a circuit for simulation. In the NDL net list format, any net renaming or bus merge-demerge connectors produce equivalence statements in the NDL net list. Modules with wide busses and bit rearrangement can sometimes contain five times as many equivalence statements as instances. A translator can execute an equivalence class computation and select a single base name that is then used in the destination format. The computation can be executed more quickly when processing individual models during net list translation than it can during circuit expansion. If all synonym names must be preserved in the flattened circuit, base name selection is not applicable.

9. Conclusions and Open Problems

Even though the argument for net list translators seems compelling, no design method can be adopted from arguments alone. The acceptance must depend on experience in design laboratories. Net list translators deserve further study and development to determine their actual utility. Meta level design methodology problems need to be studied. Many tool developers as opposed to framework developers believe that a new tool should have full flexibility in interface command choice, net list format, and functionality. They believe the extra design flexibility allows more rapid tool growth. They would claim tool growth depends on design form following function and would point out the Unix growth pattern successes. This view is contrary to nearly everything published in recent CAD literature (see [3][4] for example). Studies that explain the discrepancy and determine which is superior would be interesting. The argument presented in this paper can be viewed as an argument against design standards. Since standards are obviously valuable in hardware interfaces, standard applicability guidelines would be useful. Finally, an argument against the whole idea of tool development was expressed at the 1989 IFIP conference [24]. Development as tool building as opposed to theory development and scientific problem solving was repeatedly criticized. It is not obvious to me what the alternative to building tools would be, but the anti tool building argument seems worth pursuing.

10. References

1. Alhnan, E. An Introduction to the Source Code Control System. Unix Berkeley 4.1BSD System Documentation, UC Berkeley, 1980.

2. Bell Telephone Laboratories, *UNIX Programmer's Manual.* Holt Rinehart, 1983, Vol. 1-5.

3. Daniell, J., and Director, S. W. An object oriented approach to CAD tool control within a design framework. *Proceedings 26th Design Automation Conference,* 1989, 197-202.

4. Dewey, A., and Gadient, A. VHDL Motivation. *IEEE Design and Test of Computers,* 5, 2(April 1986).

5. Diss, W. C. Circuit compilers don't have to be slow. *Proceedings 25th Design Automation Conference,* 1988, 622-627.

6. Electronic Industry Association. EDIF Electronic Design Interchange Format Version 2 0 0. Technical Report, 1987.

7. The IEEE, *IEEE Standard VHDL Reference Manual,* New York, 1987.

8. Feldman, S. I. MAKE - A Program for Maintaining Computer Programs. *Software Practice and Experience,* 9, 4(April 1979), 256-265.

9. Gajski, D., et. al. Why can't we agree on a single standard? (panel), *Proceedings 26th Design Automation Conference,* 1989,49.

10. Genrad Corporation, Hilo III Hardware Description Language Reference Manual. 1985.

11. Hardwick, M., and Yakoob, N. Using a database and UNIX to author CAD applications. *Proceedings IEEE ICCAD-85,* 1985, 53-55.

12. Johnson, S. C. YACC -- Yet Another Compiler Compiler. Computer Science Technical Report 32, Bell Laboratories, Murray Hill, July 1975.

13. Johnson, S. C. Lint, a C Program Checker. Computer Science Technical Report 67, Bell Laboratories, Murray Hill, 1977.

14. Jones, L. G. Fast online/offline netlist compilation of hierarchical schematics. *Proceedings 26th Design Automation Conference,* 1989, 822-825.

15. Lesk, M. E. Lex -- A Lexical Analyzer Generator. Computer Science Technical Report 39, Bell Laboratories, Murray Hill, October 1975.

16. Lipsetl, R., Marshmer, E., and Shahdad, M. VHDL - The Language. *IEEE Design and Test,* 3, 2(April 1986).

17. LSI Logic Corporation, NDL Reference Manual. 1987.

18. McWilliams, T. M., and Widdoes, L. C. SCALD: Structured computer aided logic design. *Proceedings 15th Design Automation Conference,* 1978, 271-277.

19. Mehmood, Z., Singer, D., Singhal, A., and Wu, K. A design data management system for CAD. *Proceedings IEEE ICCAD-87,* 1987, 220-223.

20. Meyer S. A data structure for circuit net lists. *Proceedings 25th Design Automation Conference,* 1988, 613-616.

21. Morgan, H. L. Spelling Correction in System Programs. *CACM,* 13, 2(Feb. 1970), 90-94.

22. Rappaport, A. Data Transfer and Chip Layout Ready IC for Production. *EDN Magazine,* 3 May 1984, 280-288.

23. Ritchie, D. M., and Thompson, K. L. The Unix Time Sharing System. *CACM,* 17, 7 (July 1974).

24. Ritter, G. (ed.). Foundations of Software Engineering - the Silver Bullet (panel), *Information Processing 89,* North Holland, Amsterdam, 1989, 953-956.

25. Roberts, M. CEDIF: A data driven EDIF reader. *Proceedings 26th Design Automation Conference,* 1989, 818-821.

26. Rochkind, M. J. The Source Code Control System. *IEEE Trans. on Software Engineering*, 1, 12 (December 1975).

27. Rosenberg, J. The making of Vivid. *Proceedings 24th Design Automation Conference,* 1987, 74-81.

28. Szygenda, S. A. TEGAS 2 - Anatomy of general purpose test generation and simulation system. *Proceedings 9th Design Automation Conference,* 1972, 116-127.

29. Wagner, R. A., and Fischer, M. J. The String-to-String Correction Problem. *JACM,* 21, 1(January 1974), 168-173.

30. Yasuura, H., and Ishiura, N. Semantics of a hardware design language for Japanese standardization. *Proceedings 26th Design Automation Conference,* 1989, 836-839.